2

AD-A206 911

RADC-TR-87-219
Final Technical Report
September 1988

RESEARCH ON SIGNAL PROCESSING SUPERCOMPUTERS

Carnegie Mellon University

H. T. Kung

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

RADC-TR-87-219 has been reviewed and is approved for publication.

APPROVED: Richard N. Smith

RICHARD N. SMITH
Project Engineer

APPROVED: [illegible]

BRUNO BEEK
Technical Director
Directorate of Communications

FOR THE COMMANDER: [illegible]

JAMES W. HYDE III
Directorate of Plans & Programs

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-87-219
6a. NAME OF PERFORMING ORGANIZATION Carnegie Mellon University 6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (DCCD)
6c. ADDRESS *(City, State, and ZIP Code)* Department of Computer Science Pittsburgh PA 15213	7b. ADDRESS *(City, State, and ZIP Code)* Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center 8b. OFFICE SYMBOL *(If applicable)* DCCD	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-81-C-0206
8c. ADDRESS *(City, State, and ZIP Code)* Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 61102F; PROJECT NO. 2305; TASK NO J8; WORK UNIT ACCESSION NO. P8

11. TITLE *(Include Security Classification)*
RESEARCH ON SIGNAL PROCESSING SUPERCOMPUTERS

12. PERSONAL AUTHOR(S)
H. T. Kung

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Final	FROM Sep 85 TO Sep 86	September 1988	52

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	
25	02		Computing signal processing systolic parallel
17	04	01	

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Signal processing is an area where the required computational bandwidth in an application can be unbounded. Applications such as radar, sonar and communications already call for signal processing systems capable of delivering billions or tens of billions of operations per second. In developing a new signal processor to meet these requirements, it is essential to understand the underlying computational models. An ad-hoc processor development effort that is unclear on the computational models will likely be wasteful and unable to meet the long-term performance goal. Fortunately, because the control in signal processing is typically data-independent, computational models in this area can be relatively simple. Based on the study performed under this contract, this report describes some important computational models for parallel signal processing, and illustrates how the Warp machine developed by Carnegie Mellon supports these models. The viewpoint expressed in this report may provide some useful insights into the development of the next-generation signal processing supercomputers.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Richard N. Smith	22b. TELEPHONE *(Include Area Code)* (315) 330-3224 — 22c. OFFICE SYMBOL RADC (DCCD)

DD Form 1473, JUN 86 — *Previous editions are obsolete.* — SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

1. Background

There are serious problems with current processors for high-speed signal processing. To meet speed requirements in real-time applications, many companies, including AT&T, GE, Honeywell, Hughes, IBM, TRW, TI, and Westinghouse, have developed their own versions of programmable signal processors. The development of these processors has been enormously expensive, and yet they are far from ideal. These processors are typically programmed in assembly languages, and as a result, it is extremely difficult to develop and maintain software for them. Moreover, these machines rely heavily on custom CPUs and special-purpose hardware to obtain their performance. It is not clear at all how these systems can be scaled up to provide another order of magnitude improvement in performance without incurring a huge cost.

For a contrast, consider the community of scientific computing, which also deals with computationally demanding applications. That community enjoys access to general purpose supercomputers built by companies such as CDC, CRAY and NEC, and even smaller companies such as Convex. It has not been as necessary for researchers in scientific computing to build their own high-performance computers, as it has been for professionals in high-speed signal processing. One can easily see that computing needs for scientific computing have been more cost-effectively met than those for high-speed signal processing.

There could be many reasons for this phenomenon. High-speed signal processing machines are often embedded in larger systems, so in this case special designs are needed to deal with various interfaces to the external world. It may be also due to the fact that applications usually impose stringent power and size limitations on signal processors. However, we believe that there are other, subtle reasons. Consider for example the impact of programming languages being used in these two communities. The scientific community uses FORTRAN throughout. They know very well that all they need is machines that can execute FORTRAN code efficiently and have an effective vectorizer compiler. This goal is clearly stated, and (despite FORTRAN being an old programming language) concentrated efforts in machine architectures and their implementations have been possible. The high-speed signal processing community on the other hand has mostly been programming in assembly or lower level languages; programming experiences are highly machine-dependent. Therefore, the requirements of supercomputers for signal processing have not been clear.

To solve these problems, a fundamental way is to understand the computational models that high-performance signal processors need to support. This is elaborated in the following section.

2. The Importance of Computational Models

When developing a new computer system, some models about the computations that the machine will support efficiently are always in the designer's mind. For example, a typical signal processor is optimized to execute data-independent inner loops for routines such as FFT, filtering and matrix multiplication. Computational models are more fundamental than architectures, because the former define the usage patterns of the machines from which the latter are derived. Unfortunately, these models are often not explicitly stated, because sometimes it is difficult to describe them precisely.

The importance of computational models increases for new signal processing machines that use parallelism as a mechanism for achieving additional performance. High performance execution on parallel architecture is achieved by having the programmer or compiler organize the computation so that many tasks can be performed concurrently. The computational models are needed to give guidance on how the partitioning can be done, and how the communication cost between the processors can be minimized. Without the computational models, it would be very difficult to manage the kind of complexity due to the parallelism.

More importantly, computational models provide necessary insights about the design of high-level programming languages to support parallel computations. The most difficult part of the design of such a language is on the inter-processor communication, and this can be done properly only if the computational models have been clearly defined. Similarly the models give the hardware requirement to support efficient inter-processor communication.

Rather than proposing new architectures and discussing their computational bandwidths, in the following we give computational models for future high-performance signal processors. These models are based on our experience in projects such as Warp and *i*Warp, in design of parallel algorithms such as systolic algorithms, and in signal and image processing applications such as the autonomous land vehicle

(ALV) navigation.

We give computational models only for parallel computers using partitioned, rather than, shared memory. These computers are capable of delivering very high computational throughput because all of their processors can work simultaneously on their own local memories. Almost all of the very high-performance, parallel signal processors available today, including Warp, are machines of this kind. This, we expect, will remain to be true in the foreseeable future.

It is well-known, however, that partitioned memory parallel computers are more difficult to program than shared memory ones, because users will have to manage explicitly various memories present in the system. As stated above, computational models identified in this report will help specify hardware and software tools needed to aid the programming.

3. Some Experience at Carnegie Mellon University

High-speed signal architectures have been a focus of research at Carnegie Mellon for many years. Two of our most recent efforts in this area are the Warp and *i*Warp projects.

The Warp machine is a systolic array computer of linearly connected cells, each of which is a programmable processor capable of performing 10 million floating-point operations per second (10 MFLOPS). A typical Warp array includes 10 cells, thus having a peak computation rate of 100 MFLOPS. The Warp array can be extended to include more cells to accommodate applications capable of using the increased computational bandwidth. Warp is integrated as an attached processor into a UNIX host system. Programs for Warp are written in a high-level language supported by an optimizing compiler.

The Warp system is depicted in Figure 1. The Warp array performs the computation-intensive routines such as image processing routines or matrix operations. The interface unit handles the input/output between the array and the host, and can generate addresses (Adr) and control signals for the Warp array. The host supplies data to and receives results from the array. In addition, it executes those parts of the application programs which are not mapped onto the Warp array. For example, the host may perform decision-making processes in robot navigation or evaluate convergence criteria in iterative methods for solving systems of linear equations.

The Warp array is a linear systolic array of identical cells called Warp cells, as shown in Figure 1. Data flow through the array on two communication channels (X and Y). Those addresses for cells' local memories and control signals that are generated by the interface unit propagate down the Adr channel.

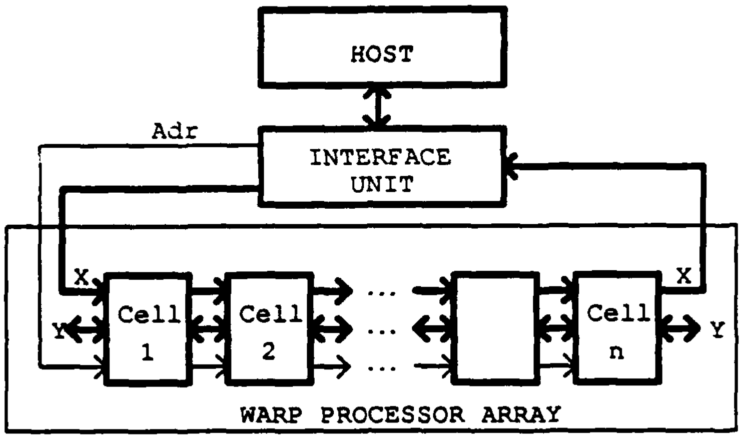

Figure 1. Warp system overview

The direction of the Y channel is statically configurable. This feature is used, for example, in algorithms that require accumulated results in the last cell to be sent back to the other cells (e.g., in back-solvers), or require local exchange of data between adjacent cells (e.g., in some implementations of numerical relaxation methods).

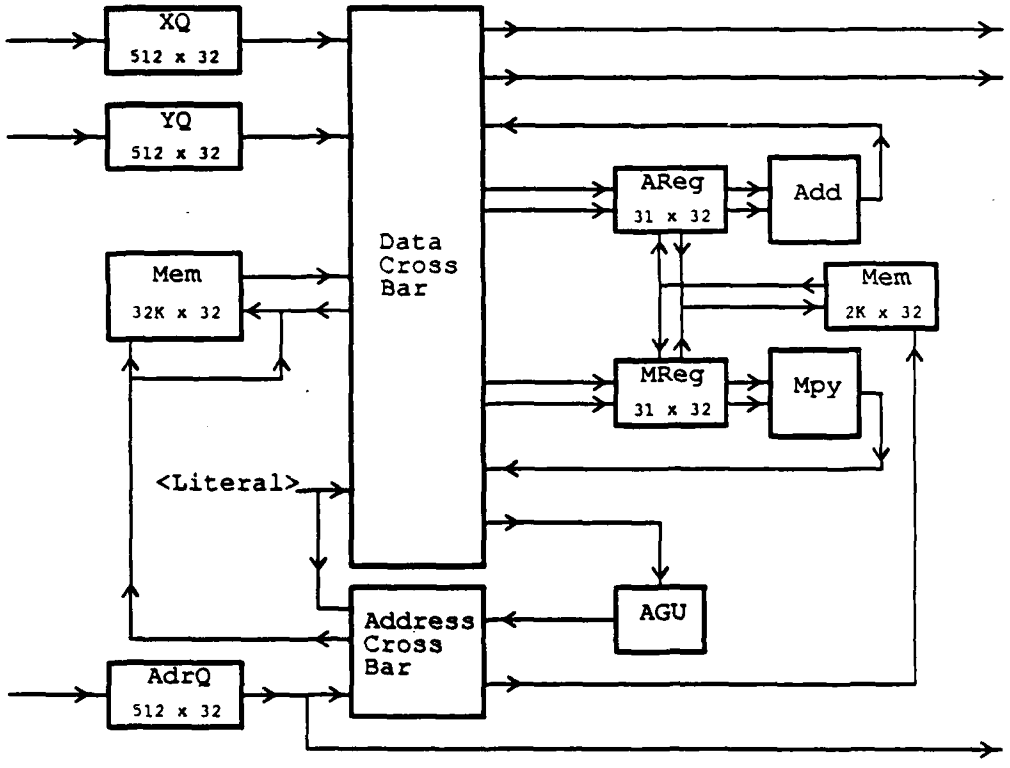

Figure 2. Warp cell data path

The first 10-cell prototype was completed in February 1986; delivery of production machines by our industrial partner (GE) started in April 1987. Extensive experimentation with both the prototype and production machines has demonstrated that the Warp architecture is effective in the application domain of robot navigation, as well as in other fields such as signal processing, scientific computation, and computer vision research [2, 1, 3, 5, 6, 14]. For these applications, Warp is typically several hundred times faster than a VAX 11/780 class computer.

Presently mounted inside of a robot vehicle called NAVLAB, Warp has been used in vehicle control to perform road following and obstacle avoidance. We have implemented road following using color classification, obstacle avoidance using stereo vision, obstacle avoidance using a laser range-finder, and path planning using dynamic programming. We have also implemented a significant portion (approximately 100 programs) of an image processing library on Warp [15], to support robot navigation and vision research in general.

Anticipating the future need for integrated Warp systems, we have been developing a chip with Intel Corporation, called the *i*Warp chip, since April 1986. When it becomes operational in 1989-90, the resulting *i*Warp system is expected to represent an order of magnitude improvement in cost and performance over the current Warp. Using tens of cells, the *i*Warp system will be able to deliver over a billion floating-point operations per second.

4. Some Background on Systolic Arrays

The Warp architecture evolves from many years' research in systolic arrays at Carnegie Mellon and elsewhere. It is therefore important to review the basic concept of systolic arrays.

4.1. Principle of Systolic Arrays

Systolic arrays are suited for "front-end processing" that deals with large amounts of data obtained directly from sensors. Although processing of this kind usually requires much computing power, it is highly regular and parallelizable. The systolic array architecture exploits this regularity and parallelism to meet the computation requirement with low costs.

The principle of a systolic array architecture (Figure 3) is that by replacing a single processing element (PE) with an array of processing elements, called cells, a higher computation throughput can be achieved without increasing the input/output bandwidth with the outside world [11]. The function of the memory is analogous to that of the heart; it "pulses" data through the array of cells. The crux of this approach is to ensure that once a data item is brought out from the memory it can be used effectively at each cell it passes while being "pumped" from cell to cell along the array. Being able to use each input data item a number of times is just one of the many advantages of a systolic array. Other advantages include modular expandability, simple and regular data and control flows, use of simple and uniform cells, efficient fault-tolerant schemes, and elimination of global data communication. These properties are highly desirable

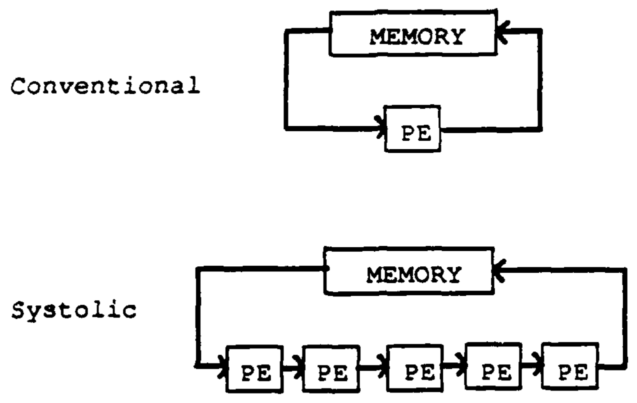

Figure 3. Processor architecture: conventional processor with one processing element (PE), and systolic array processor with an array of PEs or cells

for VLSI> (Very Large Scale Integration) implementations. Indeed the advances in VLSI technology have been a major motivation for recent interest in systolic arrays.

Systolic arrays typically call for simple and regular array interconnections between their processing elements. Many systolic algorithms have been developed on arrays depicted in Figure 4. A bibliography maintained at Carnegie Mellon lists more than 350 papers published in the past eight years on systolic arrays.

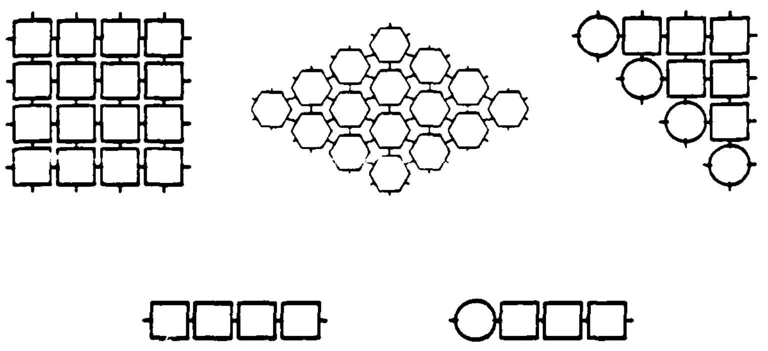

Figure 4. Typical interconnection schemes for systolic arrays

4.2. Properties of a Systolic Array Machine

We summarize some of the typical properties of a systolic array machine.

S1. The systolic array is attached to a *host*, which represents the "outside world" that supplies data and receives results to and from the array, respectively. The host may also control the array. This differs from a traditional "cellular automaton," which is assumed to be self-sufficient for the entire computation.

S2. The machine achieves its efficiency by a *careful mapping* of computation onto the systolic array.

(a) The mapping requires only simple and regular inter-cell communication for the array.

(b) Only boundary cells communicate with the outside world so the array's external I/O bandwidth is minimized.

This is unlike data flow computers where the mapping is done dynamically in an unpredictable manner.

S3. Cells of the systolic array are optimized for inter-cell communication, so data can efficiently flow through the array as they are being processed.

(a) Each cell has sufficient I/O bandwidth for efficient implementation of very *fine-grain* parallelism (e.g., only one or two arithmetic operations performed for each I/O operation).

(b) Systolic communication: each cell can operate directly on data residing at the cell's input queues and move computed results directly to the cell's output queue. Therefore it may not be necessary to store incoming or outgoing data in the cell's local memory.

This differs from a typical message passing, distributed memory parallel computer such as a hypercube.

4.3. Why the Warp Project?

A systolic array can implement a special-purpose processor, or a programmable processor. For special-purpose implementation, the systolic array is justifiable by a *predetermined* set of application tasks. Clever systolic algorithm design and highly optimized implementation, possibly with custom-made I/O devices, for the tasks are the name of the game. Tools for fast turnaround implementation are sometimes important.

When implementing a programmable systolic array processor considerations span more dimensionalities, and issues are in general more complex. One must develop programming models and support, handle the I/O with a general purpose host computer, and compete with many other programmable parallel computers. It is not simple to strike a balance between competing design goals such as high performance, low cost, and high degree of programmability; only extensive experiments will provide the necessary insights.

The objective of the Warp project is to explore the design space of *high-performance* and yet highly *programmable* systolic array machines, and prove that the resulting architecture will be cost-effective when compared to other parallel architectures. Programmability here does not mean merely that the hardware is flexible so that it can be reconfigured to perform a variety of tasks. We need to show that efficient and effective programming tools can be developed on the system, and with these tools lots of algorithms and applications can be implemented at relatively low cost.

5. Computational Models Supported by Warp

The current Warp system supports the following computational models for linear, or 1-dimensional (1D), processor arrays:

1. pipelining;
2. data partitioning;
3. recursive computation;
4. domain decomposition;
5. divide-and-conquer;
6. multi-function pipelining.

In the following we briefly describe these models. In the discussion cells in the 1D processor array are named as cell 1, cell 2, $\cdots$, cell N from left to right.

5.1. Pipelining

In this model, typical of systolic processing, the algorithm is partitioned among many Warp cells, where each cell does one stage of the processing. More precisely the computation for each output is partitioned into a sequence of identical stages, and cell i is responsible for stage i. A characteristic of this model is that cell i+1 uses computed results of cell i, as depicted in Figure 5.

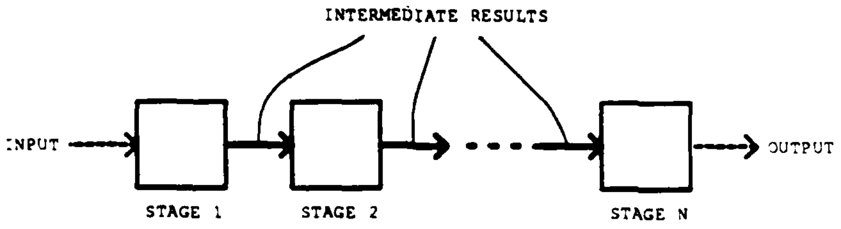

Figure 5. Pipelining model

Thus during the computation cell i+1 cannot start its operation until cell i completes at least a stage of computation. Intermediate results move from left to right, and final results emerge from the right-most cell. The sequence of computation in computing each output is exactly the same as that for the sequential one.

The Warp array's high inter-cell communication bandwidth and effectiveness in handling fine-grain parallelism make it possible to use this model. For some algorithms, this is the only method of achieving parallelism that is possible.

A simple example of the use of pipelining is the solution of elliptic partial differential equations using successive over-relaxation [18]. Consider the following equation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y).$$

The system is solved by repeatedly combining the current values of u on a 2-dimensional grid using the following recurrence:

$$u'_{i,j} = (1-\omega)\, u_{i,j} + \omega \frac{f_{i,j} + u_{i,j-1} + u_{i,j+1} + u_{i+1,j} + u_{i-1,j}}{4}, \quad \text{where } \omega \text{ is a constant parameter.}$$

In the Warp implementation, each cell is responsible for one relaxation, as expressed by the above equation. In raster order, each cell receives inputs from the preceding cell, performs its relaxation step, and outputs the results to the next cell. While a cell is performing the k^{th} relaxation step on row i, the preceding and next cells perform the $k-1^{st}$ and $k+1^{st}$ relaxation steps on rows $i+2$ and $i-2$, respectively. Thus, in one pass of the u values through the 10-cell Warp array, the above recurrence is applied ten times. This process is repeated, under control of the external host, until convergence is achieved.

The Warp implementation of FFT also uses this pipelining model [12, 14].

5.2. Data Partitioning

In this model, data are partitioned across the cells and each output is computed entirely within a cell. That is, the entire computation for each output is done *locally* at a cell. The input required by the computation of a cell is shifted in via the cells to the left. The output produced by a cell is shifted out via the cells to the right. This computation model is depicted by Figure 6, in which dotted arrows denote the shift-in and shift-out paths for input and output, respectively.

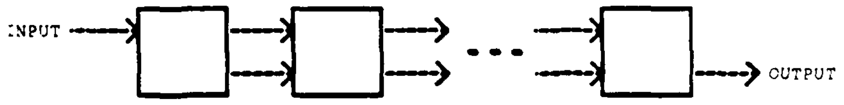

Figure 6. Data partitioning model

Various partitioning schemes can be used to assign computations to cells for the data partitioning computation model. Most of the schemes are based on the partitioning of the input or output data set [14].

5.3. Recursive Computation

The above models involve data flowing in one direction, that is, from left to right. However, bi-directional data flows are often used for computations where previously computed results are needed to compute future results. By flowing results that were previously computed against the flow of intermediate results that are currently being computed, recursive computations can be implemented. The important feature of the recursive computation model is the presence of these bi-directional data flows over the 1D array, as illustrated by Figure 7. Examples of recursive computations that have been implemented on 1D arrays with bi-directional data flows include recursive filtering [10], solution of triangular linear systems [13], and QR-decomposition [8].

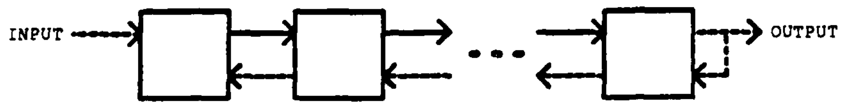

Figure 7. Recursive computation model

5.4. Domain Decomposition

The domain decomposition model arises when a problem domain (such as the grid space used in a finite difference or finite element modelling) is decomposed so that each cell handles a subdomain. This model is like the local computation model where each output is computed entirely by a single cell. However, once in a while bi-directional exchanges of information between neighboring cells are needed. The exchanges of information are relatively infrequent; they occur only after cells have done a fairly large amount of computations locally. The information exchanged between two neighboring cells involves intermediate results computed by both cells. Figure 8 depicts the domain decomposition model. In contrast, for the recursive computation model of Figure 7, bi-direction exchanges of information are relatively frequent, and each right-to-left arrow carries previously computed results by the array rather than intermediate results computed by the sending cell.

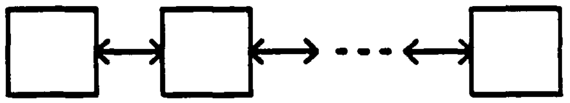

Figure 8. Domain decomposition model

There are many computations that can be conveniently carried out using the domain decomposition model. Numerical simulations of properties of a physical object, by either PDE or Monte Carlo, can be partitioned along the physical space. A large file can be sorted on a 1D array by using the bi-directional communication to merge sublists sorted by individual cells. The merging can be done in a manner similar to that used in the odd-even transposition sort, involving only nearest neighbor communications [4]. Labelling of connected components in an image can be done by using the bi-directional communication to merge labels of subimages computed by individual cells [14].

5.5. Multi-function Pipelining

A single computation may involve a series of subcomputations each executing a *different* function. If the different function stages can be chained together on the 1D array, then a one-pass execution of the entire computation would be possible. This is the basic idea of the multi-function pipelining model [6]. In this model, the 1D array is a pipeline of several groups, each consisting of a number of cells devoted to a different function. The number of cells in each group can be adjusted so that every group will take about the same time, in order to maximize the pipeline throughput.

Figure 9 illustrates the use of the multi-function pipelining model to implement the geometry system portion of 3-D computer graphics. The first cell performs the matrix multiplications, the next three cells do clipping, and the last cell does the scaling operation. Three cells are devoted to clipping as it requires more arithmetic operations than either matrix multiplication or scaling [9].

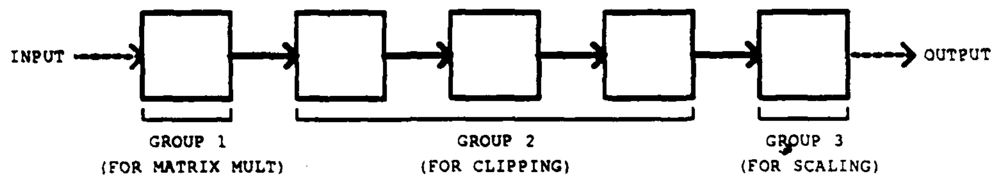

Figure 9. Multi-function pipelining model to implement a geometry system

The data rate and format of the input to a group may not be compatible to those of the output from the preceding group. In this case a buffering capability is needed at either end of a group.

Figure 10 depicts another example of multi-function pipeline. This is a laser radar simulation that we have recently implemented on Warp:

Group 1: Perform 1024-point complex FFT using 10 cells, then partition the FFT output sequence into 30 overlapped 256-element subsequences.

Group 2: For each of the 30 256-element subsequence, perform the following operations.

- Cell 1: Multiply each element by a complex number (weight).
- Cells 2-9: Perform 256-point complex inverse FFT.
- Cell 10: Compute the amplitude of each of the 256 outputs.

Group 3: Threshold the resulting 30×256 image using 3×3 windows.

The figure shows that all the operations in the three groups are performed in one pass on a linear array.

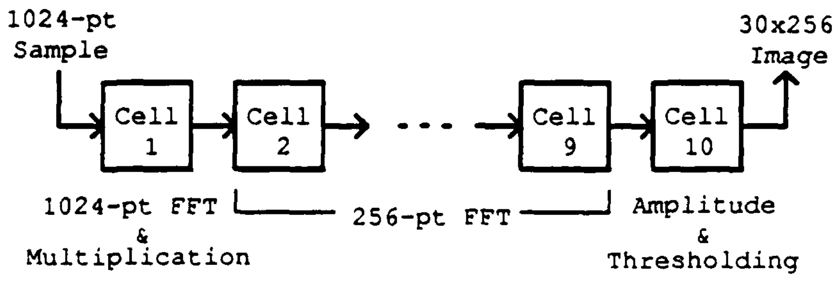

Figure 10. Radar processing on Warp

In summary, the multi-function model differs from the pipelining model described earlier in that cells are now allowed to perform different functions. This flexibility in the usage offers the opportunity of effectively using a large number of cells in a 1D array.

6. Computational Models for 2-D Arrays

This section considers 2-dimensional (2-D) processor arrays, as illustrated by Figure 11.

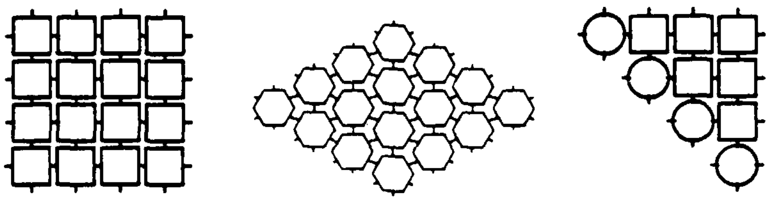

Figure 11. Examples of 2-D processor arrays

We have identified the following important computational models for 2-D arrays:

1. pipeline;
2. local computation;
3. recursive computation;
4. domain decomposition; and
5. divide-and-conquer.

These models are straightforward extensions of the corresponding ones for 1-D arrays, and thus we will discuss them only briefly.

It is possible to define query processing, multi-function pipeline and task queue models for 2-D arrays. However, they do not seem to lead to more useful applications than their counterparts for 1-D arrays.

6.1. Pipeline Model for 2-D Arrays

In the pipeline model, the 2-D array is a "wide" pipeline where each stage may consist of more than one cell. During the computation, cells in one stage send intermediate results, that they have computed, to their nearest neighboring cells in the next stage. Examples are 2-D systolic arrays for matrix multiplication [13, 17] and dynamic programming [7].

6.2. Local Computation Model for 2-D Arrays

The characteristic of the local computation model is that the computation for each output is computed entirely within a cell. An example is the matrix multiplication scheme where terms in the product matrix are accumulated locally at individual cells.

6.3. Recursive Computation Model for 2-D Arrays

In the recursive computation model, previously computed results are fed back into the array to interact with other intermediate results. An example is the LU decomposition of banded matrices [13], where previously computed results flow back to the array in two directions to meet intermediate results that flow in yet another direction.

6.4. Domain Decomposition Model for 2-D Arrays

Physical problems can often be decomposed naturally over a 2-D processor array. Each cell performs computations associated with the assigned region. When one stage of the local computation is completed, the cell communicates with its nearest neighbors to update the values on the boundary of the region.

6.5. Divide-and-conquer Model for 2-D Arrays

Both the bitonic sort and merge sort are recursive sorting methods that can be implemented on a 2-D array [16].

7. Computational Model for Heterogeneous Machines

Figure 12 indicates various tasks involved in an ALV road following application. Tasks such as road predictor and finder are well-suited to special-purpose machines employing, say, 1D or 2D processor arrays. The computational models, described earlier in this report, are useful for devising computational schemes for these individual tasks. However, to get the next level of performance we need to explore the fact that many of the tasks such as landmark recognition and road finding can operate in parallel, possibly on a variety of machines. In this section we discuss this task-level computational model for heterogeneous machines.

Driven by application needs, heterogeneous parallel computers have become increasingly common, and will represent an important trend for next-generation signal processing supercomputers. For example, the DARPA ADRIES image analysis project uses a system that integrates a 16-node Butterfly and a Symbolics 3670 for symbolic processing, as well as a Star 100 array processor, a Warp and a 128-node Butterfly for numeric processing. These processors are configured around a high-bandwidth Aptec bus, which also interfaces to a high-bandwidth disk system via a VAX 11/750.

Another example of a heterogeneous parallel computer is the current Warp machine itself. The system has a general purpose workstation and standalone MC68020 processors, in addition to the Warp array. Run-time software is provided to make these components work in parallel, and to handle various functions of the machine. Moreover, Warp is an open system in the sense that special interfaces can be added to the machine in the future to fulfill individual application needs. In fact, the design of interfaces to the Butterfly machine, the Aptec bus and a high-speed digitizer has already started.

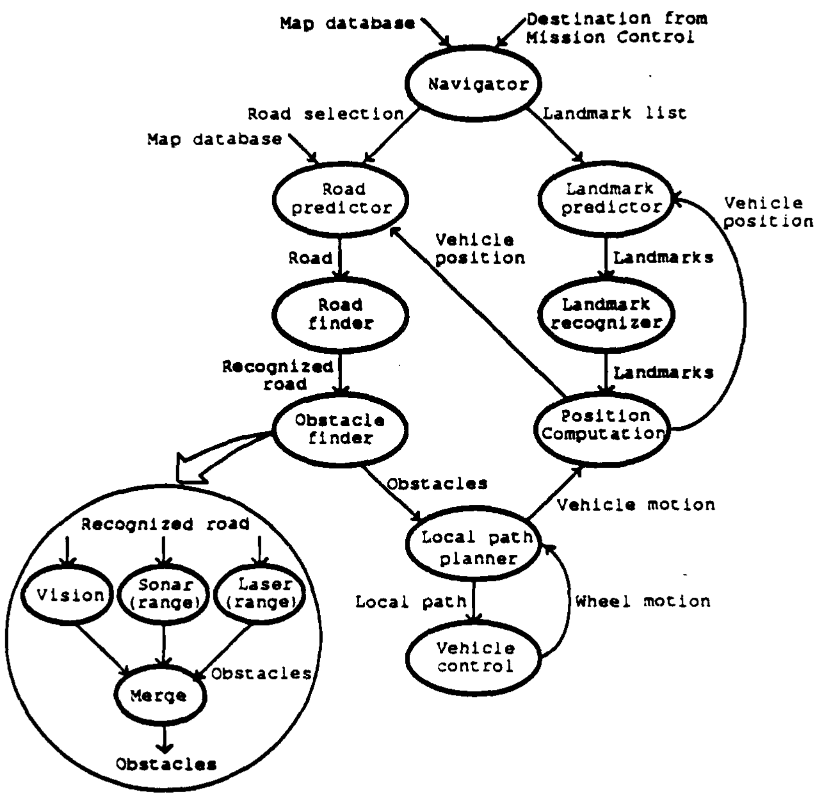

Figure 12. A task-level program for the ALV road-following

For these systems, local area networks such as Ethernet can provide the flexibility but not the required speed. To meet the speed requirement special hardware and software means are often used. However, the integration structures such as those used in the current Warp system or in the ADRIES system are ad-hoc. They do not support a unified programming environment and cannot be expanded or modified easily.

A high-performance common framework for integration purposes is crucial for open system architectures as well as for exploiting task level parallelism. The framework should support efficient use of the individual processors, and allow the user to schedule them easily.

Figure 13 depicts the configuration of a heterogeneous system. The system consists of one or more Warp arrays, several general purpose processors, sensors, and an *i*Warp array when it becomes available.

In programming the heterogeneous machine, we can use existing programming methods to program the individual processors, but we need new computational models to make the heterogeneous processors in the system work together in parallel at the task level. In the following we describe a task-level model for exploiting the task level parallelism.

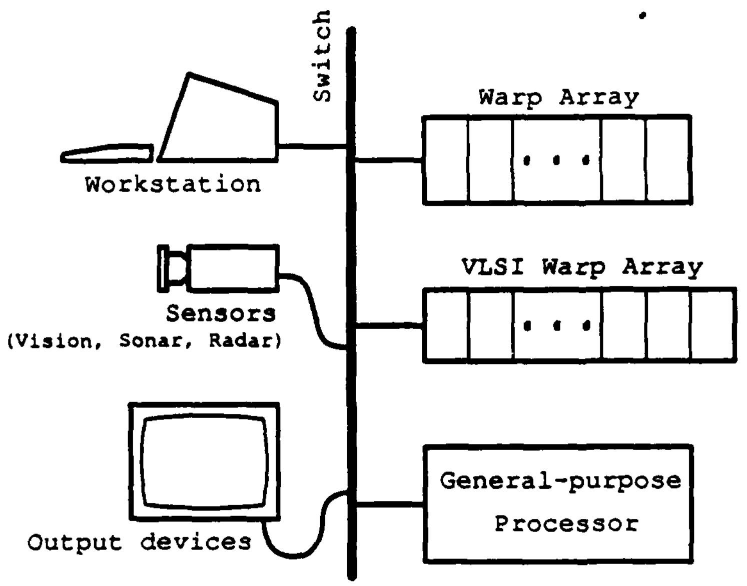

Figure 13. Configuration of a heterogeneous machine

An application program is viewed as a collection of coarse-grain, asynchronous, cooperating tasks, as depicted by Figure 14.

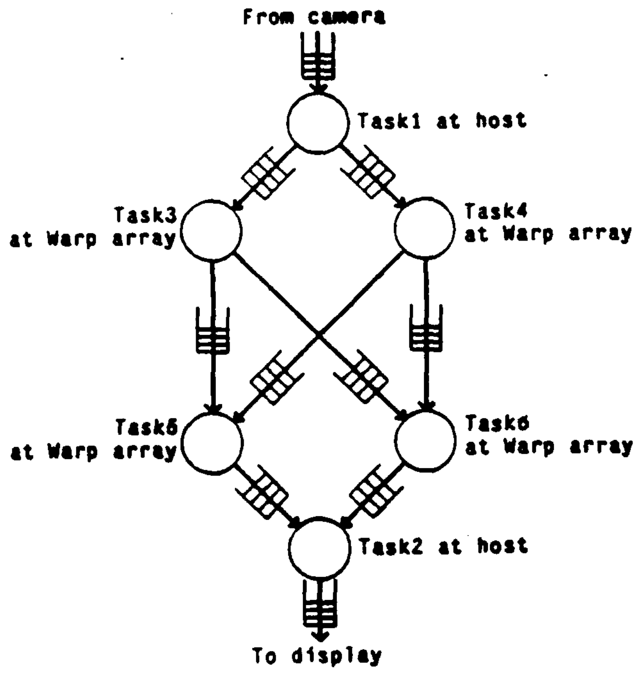

Figure 14. Program illustration using the task-level model

A task usually depends on other tasks to provide its input data, and produces output for consumption by yet other tasks. Input and output data queues can be used between tasks to smooth the data flow. Each task executes on a single special- or general-purpose processor, specified by the programmer. In general, a number of tasks within a program may execute concurrently on different processors, subject to data-dependency constraints.

The programmer can specify an input condition to trigger the execution of a task as soon as the condition is satisfied. A condition may be the minimum amount of data needed in an input queue, or the existence of certain kind of data in the queue. For instance, a landmark recognition task can start as soon as some distinguished sign appears in the image.

The programmer explicitly associates each task with a set of processors, any of which is capable of its execution. It will be up to a run-time scheduler to determine the particular processor on which the execution of the task is to be scheduled.

8. Conclusions

Computational models for 1-D and 2-D processor arrays are useful for front-end processing that deals with data directly from the sensors. The task-level model is suited to back-end processing that deals with reasoning. The ultimate signal processing supercomputer should be able to utilize the task-level parallelism provided by the task-level model, and the fine-grain parallelism provided by the computational models for 1-D and 2-D arrays.

References

1. Annaratone, M., Bitz, F., Clune, E., Kung, H. T., Maulik, P., Ribas, H., Tseng, P. and Webb, J. Applications and Algorithm Partitioning on Warp. COMPCON Spring '87, IEEE Computer Society, 1987, pp. 272-275.

2. Annaratone, M., Arnould, E., Kung, H.T. and Menzilcioglu, O. Using Warp as a Supercomputer in Signal Processing. Proceedings of ICASSP 86, IEEE, 1986, pp. 2895-2898.

3. Annaratone, M., Bitz, F., Deutch, J., Hamey, L., Kung, H. T., Maulik, P., Ribas, H., Tseng, P. and Webb, J. Applications Experience on Warp. Proceedings of the 1987 National Computer Conference, AFIPS, 1987, pp. 149-158.

4. Baudet, G. and Stevenson, D. " Optimal Sorting Algorithms for Parallel Computers". *IEEE Transactions on Computers C-27*, 1 (January 1978), 84-87.

5. Clune, E., Crisman, J. D., Klinker, G. J., and Webb, J. A. Implementation and Performance of a Complex Vision System on a Systolic Array Machine. Tech. Rept. CMU-RI-TR-87-16, Robotics Institute, Carnegie Mellon University, 1987.

6. Gross, T., Kung, H.T., Lam, M. and Webb, J. Warp as a Machine for Low-level Vision. Proceedings of 1985 IEEE International Conference on Robotics and Automation, March, 1985, pp. 790-800.

7. Guibas, L.J., Kung, H.T. and Thompson, C.D. Direct VLSI Implementation of Combinatorial Algorithms. Proceedings of Conference on Very Large Scale Integration: Architecture, Design, Fabrication, California Institute of Technology, Jan., 1979, pp. 509-525.

8. Heller, D.E. and Ipsen, I.C.F. Systolic Networks for Orthogonal Equivalence Transformations and Their Applications. Proceedings of Conference on Advanced Research in VLSI, Massachusetts Institute of Technology, Cambridge, Massachusetts, January, 1982, pp. 113-122.

9. Hsu, F.H., Kung, H.T., Nishizawa, T. and Sussman, A. Architecture of the Link and Interconnection Chip. Proceedings of 1985 Chapel Hill Conference on VLSI, Computer Science Department, The University of North Carolina, May, 1985, pp. 186-195.

10. Kung, H.T. Let's Design Algorithms for VLSI Systems. Proceedings of Conference on Very Large Scale Integration: Architecture, Design, Fabrication, California Institute of Technology, January, 1979, pp. 65-90. Also available as a CMU Computer Science Department technical report, September 1979..

11. Kung, H.T. "Why Systolic Architectures?". *Computer Magazine 15*, 1 (Jan. 1982), 37-46.

12. Kung, H.T. Systolic Algorithms for the CMU Warp Processor. Proceedings of the Seventh International Conference on Pattern Recognition, International Association for Pattern Recognition, 1984, pp. 570-577.

13. Kung, H.T. and Leiserson, C.E. Systolic Arrays (for VLSI). Sparse Matrix Proceedings 1978, Society for Industrial and Applied Mathematics, 1979, pp. 256-282.

14. Kung, H. T. and Webb, J. A. "Mapping Image Processing Operations onto a Linear Systolic Machine". *Distributed Computing 1*, 4 (1986), 246-257.

15. Electrotechnical Laboratory. *SPIDER (Subroutine Package for Image Data Enhancement and Recognition).* Joint System Development Corp., Tokyo, Japan, 1983.

16. Thompson, C.D. and Kung, H.T. "Sorting on a Mesh-Connected Parallel Computer". *Communications of the ACM 20*, 4 (April 1977), 263-271.

17. Weiser, U. and Davis, A. A Wavefront Notation Tool for VLSI Array Design. VLSI Systems and Computations, Computer Science Department, Carnegie-Mellon University, October 1981, pp. 226-234.

18. Young, D.. *Iterative Solution of Large Linear Systems*. Academic Press, New York, 1971.

Appendix: Computational Models for Future Signal Processing Supercomputers (1986 IEEE Military Communications Conference, October 1986)

ABSTRACT

Future supercomputers for signal processing will exploit parallelism available at all levels of an application. To manage the complexity due to the parallelism, computational models that define the usages of the machines must first be defined. This paper gives computational models for fine-grain parallism typically available in front-end processing that deals with data directly from sensors. The paper also describes a task-level model to capture the coarse-grain parallelism typically available in the back-end processing that does reasoning on the data obtained by the front-end processing. These models reflect the experience Carnegie Mellon University has accumulated over the past several years in the development of high-performance parallel computers for signal and image processing, and their applications.

1. INTRODUCTION

High-speed signal and image processing architectures have been a focus of research at Carnegie Mellon University for many years. Two of our most recent efforts in this area are the Warp and iWarp projects.

Warp is a programmable systolic array machine designed by Carnegie Mellon [2]. The machine has an array of 10 or more linearly connected cells, each capable of performing 10 million 32-bit floating-point operations per second (10 MFLOPS). A 10-cell array can achieve a performance of 50 to 100 MFLOPS for a large variety of signal and image processing operations.

Two wire-wrap prototypes, built by Carnegie Mellon and its industrial partners—GE and Honeywell, have been operational since spring 1986. These machines are being used intensively for signal and vision processing and for scientific computing. For these computations, the new machines are typically ten to one hundred times faster than the conventional machines we have at Carnegie Mellon. GE is under contract to build eight printed circuit board versions of the machine to support research in robot navigation and image analysis where computational demands can be extremely high.

Anticipating the future need for integrated Warp systems, Carnegie Mellon and Intel have been developing a VLSI Warp chip, called the iWarp chip, since April 1986. This project is supported in part by DARPA. The resulting iWarp system is expected to represent an order of magnitude improvement in cost-performance over the current Warp. Using tens of cells the iWarp system will be able to deliver over a billion floating-point operations per second.

In this paper we consider future signal processing supercomputers that are even more powerful than the current high-performance machines such as Warp. Rather than proposing new architectures and discussing their computational bandwidths, we give computational models for these supercomputers. These models are based on our experience in projects such as Warp and iWarp, in design of parallel algorithms such as systolic algorithms, and in signal and image processing applications such as the autonomous land vehicle (ALV) navigation. Note that computational models are more fundamental than architectures, because the former define the usage patterns of the machines from which the latter are derived.

We give computational models only for parallel computers using partitioned, rather than, shared memory. These computers are capable of delivering very high computational throughput because all of their processors can work simultaneously on their own local memories. Almost all of the very high-performance, parallel signal processors available today, including Warp, are machines of this kind. This, we expect, will remain to be true in the foreseeable future.

It is well-known, however, that partitioned memory parallel computers are more difficult to program than shared memory ones, because users will have to manage explicitly various memories present in the system. Computational models identified in this paper will help specify hardware and software tools needed to aid the programming.

We have identified eight major computational models for 1-dimensional (1-D) processor arrays. These are presented in Section 4. A brief discussion on the extension of these models to 2-D processor arrays is given in Section 5. Section 6 describes a task-level model for heterogeneous machines. This model can exploit the higher-level parallelism available between the various tasks of an application. Sections 2 and 3 provide some background information on the Warp and iWarp systems.

2. OVERVIEW OF WARP

The Warp machine has three components—the Warp processor array, or simply Warp array, the interface unit, and the host, as depicted in Figure 1. We describe this machine only briefly here: more detail is available separately [2]. The Warp processor array performs the bulk of the computation. The interface unit handles the input/output between the array and the host. The host has two functions: carrying out high-level application routines and supplying data to the Warp processor array.

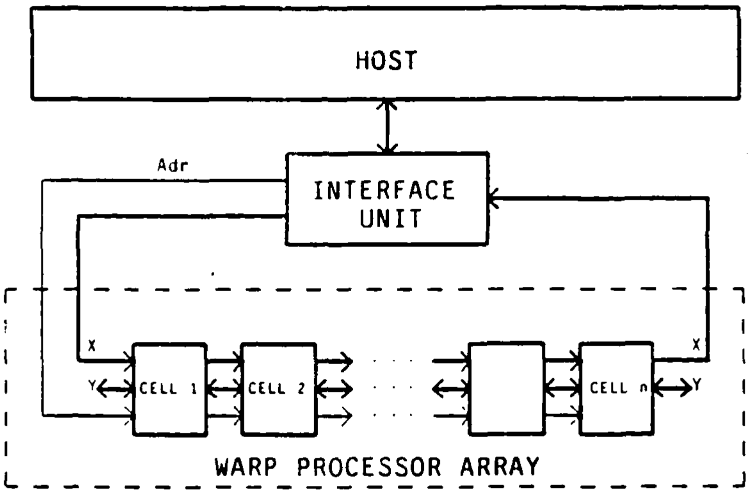

Figure 1. Warp machine overview

The Warp processor array is a programmable, linear systolic array, in which all processing elements (Warp cells) are identical. Data flow through the array on two data paths (X and Y), while addresses and systolic control signals travel on the Adr path (as shown in the Figure 1). The data path of a Warp cell is depicted in Figure 2. Each cell contains two floating-point processors: one multiplier and one ALU [23]. These are highly pipelined; they each can deliver up to 5 MFLOPS. This performance translates to a peak processing rate of 10 MFLOPS per cell or 100 MFLOPS for a 10-cell processor array. To ensure that data can be supplied at the rate they are consumed, an operand register file is dedicated to each of the arithmetic units, and a crossbar is used to support high intra-cell bandwidth. Each input path has a queue to buffer input data. A 32K-word memory is provided for resident and temporary data storage.

A feature that distinguishes the Warp cell from many other processors of similar computation power is its high I/O bandwidth—an important characteristic for systolic arrays. Each Warp cell can transfer up to 20 million words (80 Mbytes) to and from its neighboring cells per second. (In addition, 10 million 16-bit addresses can flow from one cell to the next

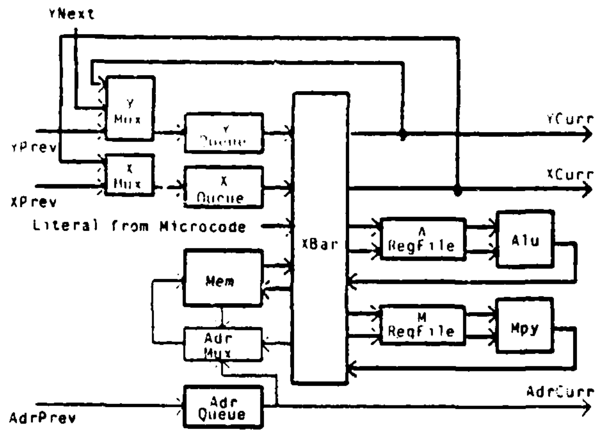

Figure 2. Warp cell data path

cell every second.) This high inter-cell communication bandwidth makes it possible to transfer large volumes of intermediate data between neighboring cells in a short time and thus supports fine grain problem decomposition.

As address patterns are typically data-independent and common to all the cells, full address generation capability is factored out from the cell architecture and provided in the interface unit. Addresses are generated by the interface unit and propagated from cell to cell (together with the control signals). However, for the printed circuit board versions of the Warp machine currently being built, each cell will also have its own address generation unit. In addition to generating addresses, the interface unit passes data and results between the host and the Warp array, possibly performing some data conversion in the process.

While achieving a high computational throughput, Warp has a high degree of programmability. Each processor is a horizontal microengine; the user has complete control over the various functional units. To help manage this fine-grain parallelism, an optimizing compiler to support a high-level programming language has been developed [6]. To the application programmer, Warp is an array of simple sequential processors, communicating asynchronously. Based on the user's program for this abstract array, the compiler generates code for the host, interface unit and Warp array automatically.

3. OVERVIEW OF IWARP

Carnegie Mellon and Intel are jointly developing a VLSI chip, called the iWarp chip, to implement an integrated version of the Warp cell. The iWarp chip is a programmable processor capable of delivering at least 16 MFLOPS.

This chip together with, say, a 64K-word local memory can form a powerful building-block cell, called iWarp cell, for a variety of processor arrays beyond the current Warp machine. This is illustrated in Figure 3.

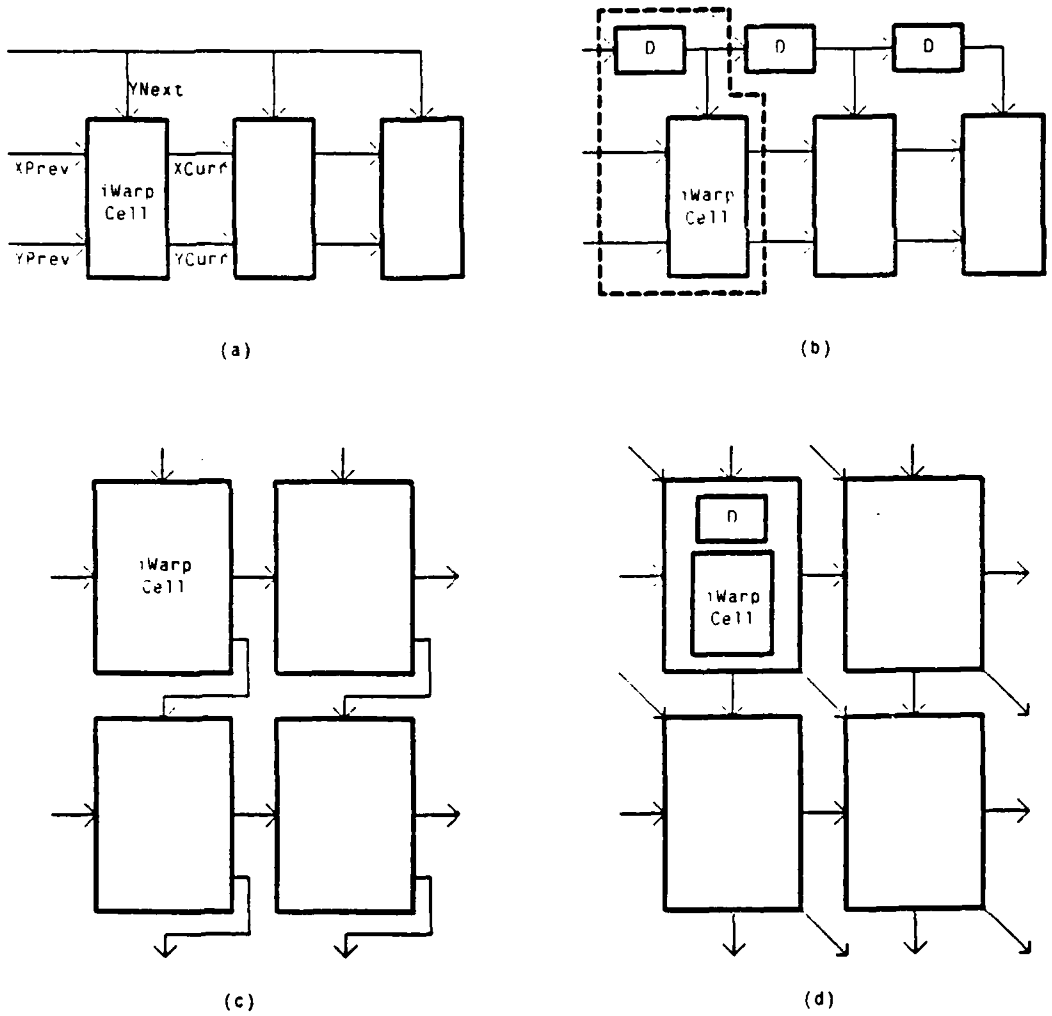

Figure 3. Processor arrays composed of iWarp cells:
(a) using the YNext input bus to receive broadcast data; (b) using the YNext input bus and delay elements, denoted by D's, to implement a new systolic pathway; (c) using the iWarp cell to implement 2-D rectangular processor arrays; and (d) using the iWarp cell augmented by a delay element to implement hexagonal processor arrays

The iWarp cell is expected to be at least 1.6 times faster than the Warp cell, and will have about a 2" x 4" footprint compared to the Warp cell whose current implementation occupies an entire 15" x 17" wire-wrap board. In addition, the iWarp cell will be able to execute application programs originally written for the Warp cell.

4. COMPUTATIONAL MODELS FOR 1-D ARRAYS

We have identified eight important computational models for 1-dimensional (1-D) processor arrays:

1. pipeline;
2. local computation;
3. recursive computation;
4. domain decomposition;
5. divide-and-conquer;
6. query processing;
7. multi-function pipeline; and
8. task queue.

The current Warp system supports the first four models, whereas the future iWarp system will support all the models. In the following we describe these models, and illustrate them by examples.

In the discussion cells in the 1-D processor array are named as cell 1, cell 2, $\cdots$, cell N from left to right.

4.1. Pipeline Model for 1-D Arrays

This is the classic systolic array model, where each output is computed across all the cells in a pipelined fashion. More precisely the computation for each output is partitioned into a sequence of identical stages, and cell i is responsible for stage i. A characteristic of this model is that cell $i+1$ uses computed results of cell i, as depicted in Figure 4.

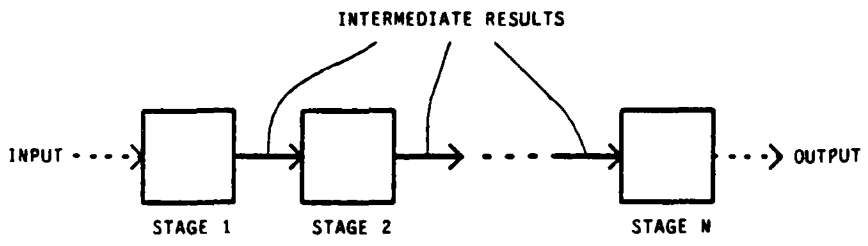

Figure 4. Pipeline model

Thus during the computation cell $i+1$ cannot start its operation until cell i completes at least a stage of computation. Intermediate results move from left to right, and final results emerge from the right-most cell. The sequence of computation in computing each output is exactly the same as that for the sequential one.

4.1.1. FFT Example

Warp implements the FFT using this pipeline model [13]. A n-point FFT, with n being a power of 2, involves $\log_2 n$ stages of $n/2$ butterfly operations, and data shufflings between any two consecutive stages. The so-called constant geometry version of the FFT algorithm allows the same data shuffling to be used for all the stages [18]. This is depicted in Figure 5 with $n = 16$. In the figure the butterfly operations are represented by circles, and number h by an edge indicates that the result associated with the edge must be multiplied by ω^h.

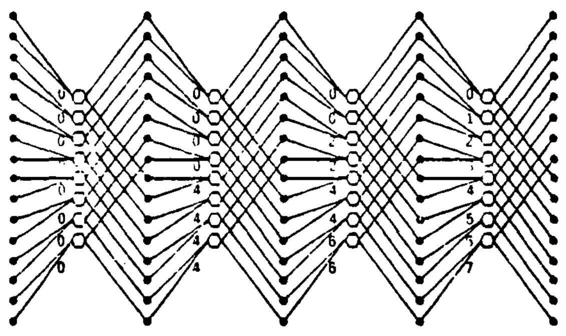

Figure 5. Constant geometry version of FFT

In the Warp array, all the butterfly operations in the i-th stage are carried out by cell i, and results are stored to the data memory of cell $i+1$. While the data memory of cell $i+1$ is being filled by the outputs of cell i, cell $i+1$ can work on the butterfly operations in the $(i+1)^{st}$ stage of another FFT problem. Note that every cell accesses its memory in a shuffled order. As the same shuffling is performed for all the stages, the interface unit can send the same address stream to all the cells. In practical applications, there are often a large number of FFTs to be processed, or there are FFT problems being continuously generated. Thus it is possible that a new FFT problem can enter the first cell, as soon as the cell becomes free. In this way all the cells of the systolic array can be kept busy all the time.

4.1.2. Relaxation Example

In some image processing algorithms, the input image is subject to multiple passes of the same operation [19, 20]. This process is called *relaxation*, in which pass $i+1$ uses the results of pass i. A natural way to implement relaxation on a 1-D processor array is to have cell i perform pass i and send results to cell $i+1$.

In a similar way we can implement many other iterative methods such as successive over relaxation (SOR), Jacobi and Gauss-Seidel methods for the solution of linear systems of equations. These methods have all been implemented on the current Warp system using the pipeline model.

4.2. Local Computation Model for 1-D Arrays

In the local computation model, each output is computed entirely within a cell. That is, the entire computation for each output is done *locally* at a cell. The input required by the computation of a cell is shifted in via the cells to the left. The output produced by a cell is shifted out via the cells to the right. The local computation model is depicted by Figure 6, in which dotted arrows denote the shift-in and shift-out paths for input and output, respectively.

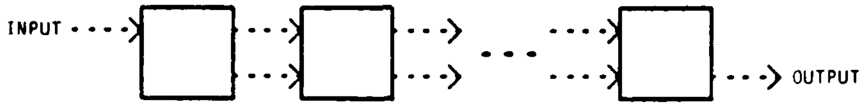

Figure 6. Local computation model

Various partitioning schemes can be used to assign computations to cells for the local computation model. Most of the schemes are based on the partitioning of the input or output set [16].

4.2.1. 2-D Convolution Example

One may implement the 2-D convolution on a 1-D array by distributing the input image evenly to all the cells prior to computation, and then during the computation having cells work independently from each other using only data local to the cells. There are many other examples based on this local computation model using only local inputs during the computation. They include the discrete cosine transform [3] and the labelled histogram computation [16].

4.2.2. Matrix Multiplication Example

Given matrices A and B, their product $A \cdot B$ can be computed using the local computation model. Prior to the computation columns of B are distributed evenly among the cells; during the computation rows of A are passed to all the cells, and columns of the product $A \cdot B$ are computed locally at cells [13]. This differs from the above 2-D convolution example where no input is shifted into a cell during the computation.

4.2.3. Dynamic Programming Example

A dynamic programming algorithm has been implemented on Warp to find the shortest paths for terrain images. In this program, cell i computes row i of the output image.

However, unlike the preceding 2-D and matrix multiplication examples, cell i uses results of cell $i-1$. This implies that cell i cannot start its computation for row i until cell $i-1$ has at least finished some of the computation for row i-1. In fact, during the computation cell i lags behind cell i-1 by two positions. That is, while cell i computes position j of row i, cell $i-1$ will be computing position $j+2$ of row $i-1$.

Other examples of pipeline with low intercell latency include 1-D convolution and polynomial evaluation at many points.

4.2.4. The Hough Transform Example

The Hough transform is a template matching algorithm originally invented to find lines in cloud chamber photographs [10] and later generalized to find arbitrary parameterized curves [4]. The algorithm works by mapping each significant pixel of an image into a set of locations in a table representing different locations in the parameter space. The mapping takes each pixel in the image into all possible combinations of parameters generating curves that pass through the image pixel.

For example, in line-finding, lines are parameterized by two values, θ and ρ. The line described by a particular pair of values of these parameters is

$$x\cos\theta + y\sin\theta = \rho.$$

Thus, for line finding, the Hough transform takes the (x,y) location of each significant pixel and, over a range of θ values, calculates the ρ value for this (x,y) using the formula above. It then increments a table at location (θ, ρ). Once the entire data set has been processed, the table is scanned and peaks are found. These peaks represent the most likely lines in the image.

The time-consuming step in this algorithm is the mapping between the image and the parameter space. This can involve floating-point computation and must be done once for each significant pixel in the image, which can be a good portion of the image. Also, the parameter space searched can be quite large, depending on its dimensionality and the granularity of the parameter search.

The Warp implementation of the Hough transform works by dividing the parameter space into different segments to a cell's memory, then allocating each segment to one Warp cell. The host preprocesses the image by selecting significant pixels and sending their locations to Warp (alternatively, these pixel locations can be generated in a pass of the image through Warp). The location of each significant pixel is sent to every cell systolically. At each cell, the segment of the Hough space that belongs to the cell is indexed by some set of parameters $p_1, p_2, \ldots, p_n$. The pixel location is fed into a formula with some particular value of the first $n-1$ parameters and the n^{th} parameter is generated. Table lookup for unary functions such as sine or cosine can be used to simplify computation of the n^{th} parameter. The table element at this location is then incremented. This process is repeated until the computation for the entire segment belonging to the cell is completed, and then the pixel location is sent on to the next cell. Thus in the steady state, all the Warp cells carry out computations simultaneously for different segments. After all the significant pixels have been sent through the Warp cell, each cell selects its significant peaks and sends them to the host where the maxima of all the peaks can be found.

Thus for this implementation each cell works on a subset of the output, that is, the parameter space. The entire input set, that is, the set of significant pixels, is passed to every cell during the computation. This is an example of the use of the local computation model based on the output partitioning scheme.

4.3. Recursive Computation Model for 1-D Arrays

All the models described so far involve data flowing in one direction, that is, from left to right. However, bi-directional data flows are often used for computations where previously computed results are needed to compute future results. By flowing results that were previously computed against the flow of intermediate results that are currently being computed, recursive computations can be implemented. The important feature of the recursive computation model is the presence of these bi-directional data flows over the 1-D array, as illustrated by Figure 7.

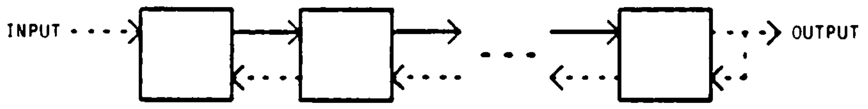

Figure 7. Recursive computation model

Examples of recursive computations that have been implemented on 1-D arrays with bi-directional data flows include recursive filtering [12], solution of triangular linear systems [14], and QR-decomposition [9].

4.4. Domain Decomposition Model for 1-D Arrays

The domain decomposition model arises when a problem domain (such as the grid space used in a finite difference or finite element modelling) is decomposed so that each cell handles a subdomain. This model is like the local computation model where each output is computed entirely by a single cell. However, once in a while bi-directional exchanges of information between neighboring cells are needed. The exchanges of information are relatively infrequent; they occur only after cells have done a fairly large amount of computations locally. The information exchanged between two neighboring cells involves intermediate results computed by both cells. Figure 8 depicts the domain decomposition model. In contrast, for the recursive computation model of Figure 7, bi-direction exchanges of information are relatively frequent, and each right-to-left arrow carries previously computed results by the array rather than intermediate results computed by the sending cell.

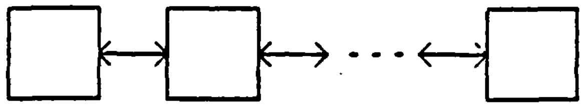

Figure 8. Domain decomposition model

There are many computations that can be conveniently carried out using the domain decomposition model. Numerical simulations of properties of a physical object, by either PDE or Monte Carlo, can be partitioned along the physical space. A large file can be sorted on a 1-D array by using the bi-directional communication to merge sublists sorted by individual cells. The merging can be done in a manner similar to that used in the odd-even transposition sort, involving only nearest neighbor communications [5]. Labelling of connected components in an image can be done by using the bi-directional communication to merge labels of subimages computed by individual cells [16].

4.5. Divide-and-conquer Model for 1-D Arrays

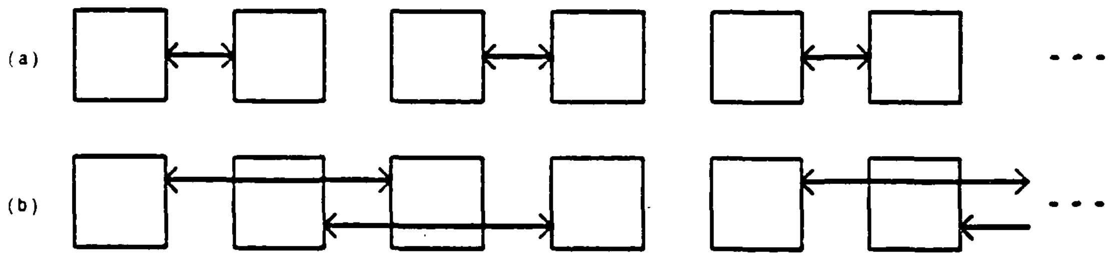

Figure 9. Divide-and-conquer model: (a) communications between cells that are 1-apart, and (b) communications between cells that are 2-apart

Divide-and-conquer is a fundamental technique in algorithm design [1]. Under this design paradigm, we solve a problem by (1) partitioning it into subproblems of nearly equal size, (2) solving all the subproblems, and (3) merging the solutions to the subproblems: this procedure is applied recursively to all the subproblems. Figure 9 illustrates the divide-and-conquer model. Each subproblem is carried out by one cell or a set of consecutive cells. When a (sub)problem is partitioned into subproblems or solutions to subproblems are merged, communications between cells that are either 1-apart, 2-apart, 4-apart, $\cdots$, or $N/2$-part take place. The 1-apart and 2-apart communications are depicted by solid arrows in the figure. A characteristic of the divide-and-conquer model is the presence of these different communications. This distinguishes the model from the local computation and domain decomposition models.

The divide-and-conquer model for example can be used in sorting, and various geometric problems such as computing convex hulls [17].

4.6. Query Processing Model for 1-D Arrays

A 1-D array can be used to process queries. One way to do this is to have the database partitioned evenly among the cells. Then queries are passed to all the cells. Every cell looks at the arriving query and outputs its reply to the query. This is illustrated by Figure 10.

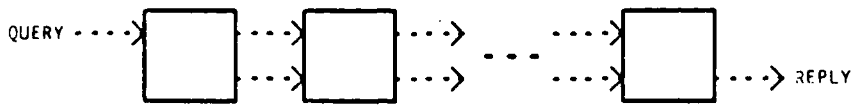

Figure 10. Query processing model

Consider for example the problem of looking for a table in an image. The particular table we are searching for is defined as having a rectangular top, which will appear as a parallelogram in the image. Initially, we do not know anything about the position of the tabie, except an upper bound on the size of its bounding square in the image. After extracting features such as lines and edges from the image, we partition it into regions whoses sizes are at least that of the bounding square for the table. We assign each region to a cell. To balance the computational load between the cells, we define the regions so that there are about the same number of features associated with each region. Regions assigned to the cells are properly overlapped to ensure that the entire table is contained in at least one region. All the cells can work in parallel on their own regions to respond to the query:

"list all sets of four lines that form a parallelogram".

Given the response to this query, the host can predict the position of other sides of the table, and produce queries such as:

"list parallel lines with a given orientation",

to find the other sides of the table.

The query processing model requires that the cells operate asynchronously, as when responding a query they may have to perform different amounts of computations and may produce variable amounts of outputs.

4.7. Multi-function Pipeline Model for 1-D Arrays

A single computation may involve a series of subcomputations each executing a *different* function. Figure 11 illustrates various functions involved in the control of an ALV road following algorithm [15].

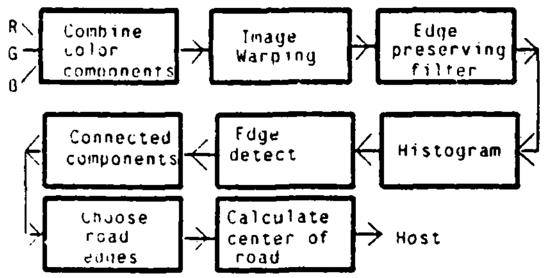

Figure 11. Multi-function pipeline in an ALV road following algorithm

If the different function stages can be chained together on the 1-D array, then a one-pass execution of the entire computation would be possible. This is the basic idea of the multi-function pipeline model [7]. In this model, the 1-D array is a pipeline of several groups, each consisting of a number of cells devoted to a different function. The number of cells in each group can be adjusted so that every group will take about the same time, in order to maximize the pipeline throughput.

Figure 12 illustrates the use of the multi-function pipeline model to implement the geometry system portion of 3-D computer graphics. The first cell performs the matrix multiplications, the next three cells do clipping, and the last cell does the scaling operation. Three cells are devoted to clipping as it requires more arithmetic operations than either matrix multiplication or scaling [11].

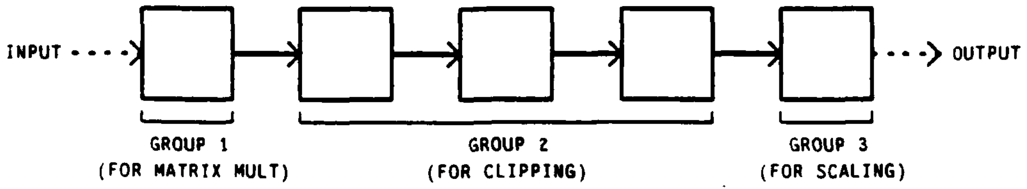

Figure 12. Multi-function pipeline model to implement a geometry system

The data rate and format of the input to a group may not be compatible to those of the output from the preceding group. In this case a buffering capability is needed at either end of a group.

In summary, the multi-function model differs from the pipeline model described earlier in that cells are now allowed to perform different functions. This flexibility in the usage offers the opportunity of effectively using a large number of cells in a 1-D array.

4.8. Task Queue Model for 1-D Arrays

For all of the preceding models, cells work together for a common task, whether they are tightly coupled (as in the pipeline model) or loosely coupled (as in the domain decomposition model). In contrast, the task queue model allows different cells to work on different tasks. More precisely, a free cell can be dynamically assigned to execute any task in a task queue maintained by the host, as depicted by Figure 13. Cells operate in a totally independent and asynchronous manner. This would allow a 1-D array to operate like an MIMD machine and support multiple users. Since the I/O for all the cells must go through the two boundary cells, to make efficient use of this model there must be many tasks each of which will do a large amount of computations per I/O operation.

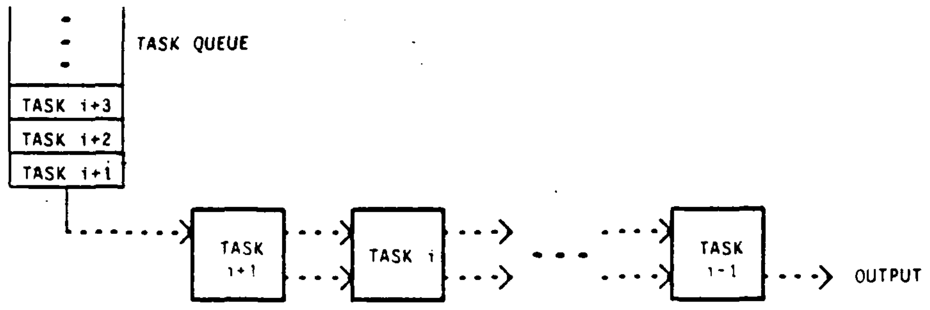

Figure 13. Task queue model

5. COMPUTATIONAL MODELS FOR 2-D ARRAYS

This section considers 2-dimensional (2-D) processor arrays, as illustrated by Figure 14.

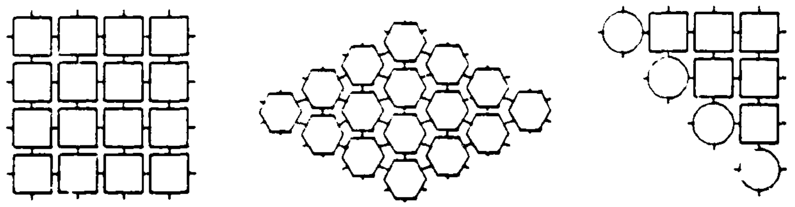

Figure 14. Examples of 2-D processor arrays

We have identified the following important computational models for 2-D arrays:

1. pipeline;

2. local computation;

3. recursive computation;

4. domain decomposition; and

5. divide-and-conquer.

These models are straightforward extensions of the corresponding ones for 1-D arrays, and thus we will discuss them only briefly.

It is possible to define query processing, multi-function pipeline and task queue models for 2-D arrays. However, they do not seem to lead to more useful applications than their counterparts for 1-D arrays.

5.1. Pipeline Model for 2-D Arrays

In the pipeline model, the 2-D array is a "wide" pipeline where each stage may consist of more than one cell. During the computation, cells in one stage send intermediate results, that they have computed, to their nearest neighboring cells in the next stage. Examples are 2-D systolic arrays for matrix multiplication [14, 22] and dynamic programming [8].

5.2. Local Computation Model for 2-D Arrays

The characteristic of the local computation model is that the computation for each output is computed entirely within a cell. An example is the matrix multiplication scheme where terms in the product matrix are accumulated locally at individual cells.

5.3. Recursive Computation Model for 2-D Arrays

In the recursive computation model, previously computed results are fed back into the array to interact with other intermediate results. An example is the LU decomposition of banded matrices [14], where previously computed results flow back to the array in two directions to meet intermediate results that flow in yet another direction.

5.4. Domain Decomposition Model for 2-D Arrays

Physical problems can often be decomposed naturally over a 2-D processor array. Each cell performs computations associated with the assigned region. When one stage of the local computation is completed, the cell communicates with its nearest neighbors to update the values on the boundary of the region.

5.5. Divide-and-conquer Model for 2-D Arrays

Both the bitonic sort and merge sort are recursive sorting methods that can be implemented on a 2-D array [21].

6. COMPUTATIONAL MODEL FOR HETEROGENEOUS MACHINES

Figure 15 indicates various tasks involved in an ALV road following application. Tasks such as road predictor and finder are well-suited to special-purpose machines employing, say, 1-D or 2-D processor arrays. The computational models, described earlier in this paper, are useful for devising computational schemes for these individual tasks. However, to get the next level of performance we need to explore the fact that many of the tasks such as landmark recognition and road finding can operate in parallel, possibly on a variety of machines. In this section we discuss this task-level computational model for heterogeneous machines.

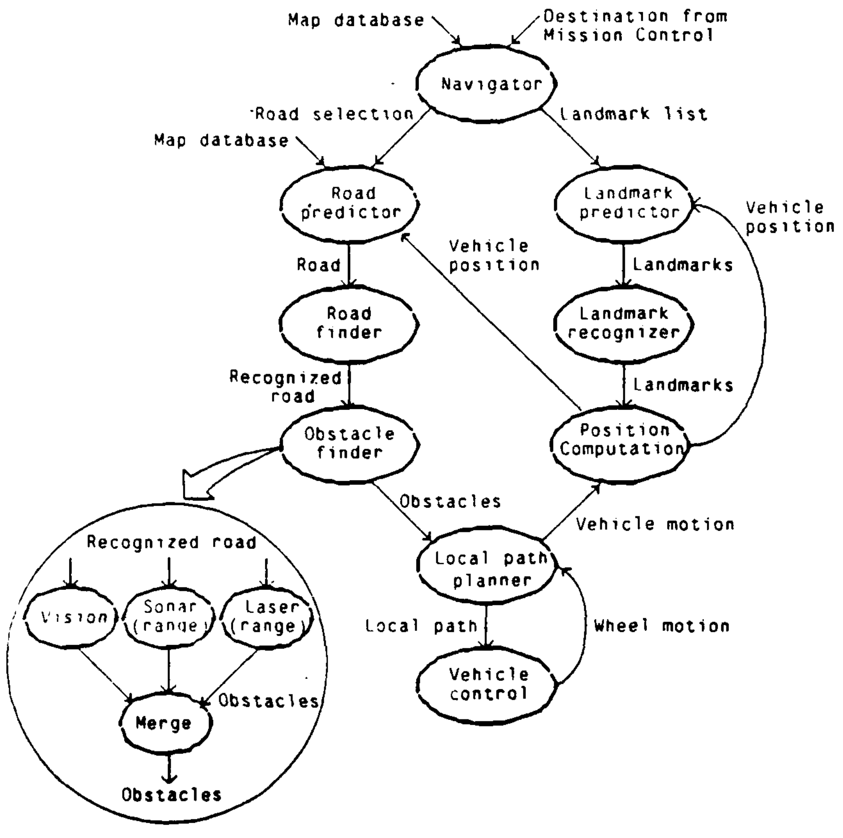

Figure 15. A task-level program for the ALV road-following

Figure 16 depicts the configuration of a heterogeneous system. The system consists of one or more Warp arrays, several general purpose processors, sensors, and a VLSI Warp array when it becomes available.

In programming the heterogeneous machine, we can use existing programming methods to program the individual processors, but we need new computational models to make the heterogeneous processors in the system work together in parallel at the task level. In the

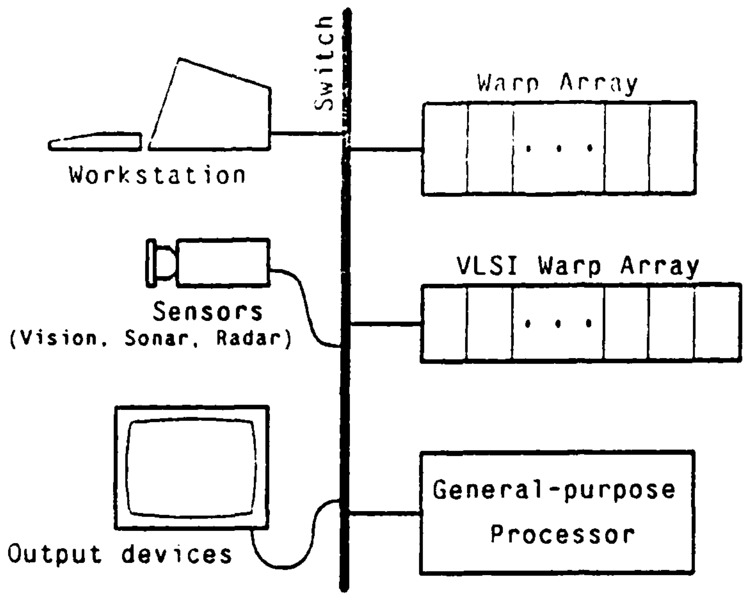

Figure 16. Configuration of a heterogeneous machine

following we describe a task-level model for exploiting the task level parallelism.

An application program is viewed as a collection of coarse-grain, asynchronous, cooperating tasks, as depicted by Figure 17.

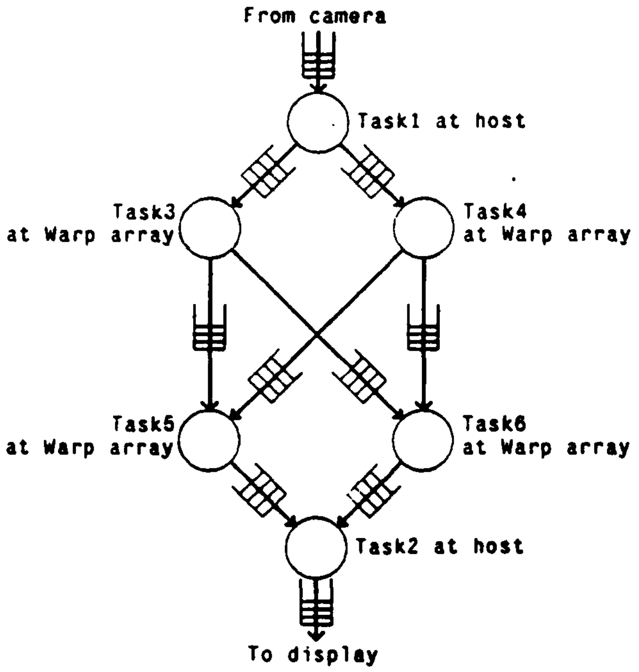

Figure 17. Program illustration using the task-level model

A task usually depends on other tasks to provide its input data, and produces output for consumption by yet other tasks. Input and output data queues can be used between tasks to smooth the data flow. Each task executes on a single special- or general-purpose processor, specified by the programmer. In general, a number of tasks within a program may execute concurrently on different processors, subject to data-dependency constraints.

The programmer can specify an input condition to trigger the execution of a task as soon as the condition is satisfied. A condition may be the minimum amount of data needed in an input queue, or the existence of certain kind of data in the queue. For instance, a landmark recognition task can start as soon as some distinguished sign appears in the image.

The programmer explicitly associates each task with a set of processors, any of which is capable of its execution. It will be up to a run-time scheduler to determine the particular processor on which the execution of the task is to be scheduled.

7. CONCLUSIONS

Computational models for 1-D and 2-D processor arrays are useful for front-end processing that deals with data directly from the sensors. The task-level model is suited to back-end processing that deals with reasoning. The ultimate signal processing supercomputer should be able to utilize the task-level parallelism provided by the task-level model, and the fine-grain parallelism provided by the computational models for 1-D and 2-D arrays.

References

[1] Aho, A., Hopcroft, J.E. and Ullman, J.D.
The Design and Analysis of Computer Algorithms.
Addison-Wesley, Reading, Massachusetts, 1975.

[2] Annaratone, M., Arnould, E., Gross, T., Kung, H.T., Lam, M., Menzilcioglu, O., Sarocky, K. and Webb, J.A.
Warp Architecture and Implementation.
In *Conference Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 346-356. June, 1986.

[3] Annaratone, M., Arnould, E., Kung, H.T. and Menzilcioglu, O.
Using Warp as a Supercomputer in Signal Processing.
In *Proceedings of ICASSP 86.* IEEE, 1986.

[4] Ballard, D. H. and Brown, D.M.
Computer Vision.
Prentice-Hall, 1982.
pp. 123-31.

[5] Baudet, G. and Stevenson, D.
Optimal Sorting Algorithms for Parallel Computers.
IEEE Transactions on Computers C-27(1):84-87, January, 1978.

[6] Gross, T. and Lam, M.
Compilation for a High-performance Systolic Array.
In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, pages 27-38. ACM SIGPlAN, June, 1986.

[7] Gross, T., Kung, H.T., Lam, M. and Webb, J.
Warp as a Machine for Low-level Vision.
In *Proceedings of 1985 IEEE International Conference on Robotics and Automation*, pages 790-800. March, 1985.

[8] Guibas, L.J., Kung, H.T. and Thompson, C.D.
Direct VLSI Implementation of Combinatorial Algorithms.
In *Proceedings of Conference on Very Large Scale Integration: Architecture, Design, Fabrication*, pages 509-525. California Institute of Technology, January, 1979.

[9] Heller, D.E. and Ipsen, I.C.F.
Systolic Networks for Orthogonal Equivalence Transformations and Their Applications.
In *Proceedings of Conference on Advanced Research in VLSI*, pages 113-122. Massachusetts Institute of Technology, Cambridge, Massachusetts, January, 1982.

[10] Hough, P. V. C.
Method and Means for Recognizing Complex Patterns.
United States Patent Number 3,069,654, December, 1962.

[11] Hsu, F.H., Kung, H.T., Nishizawa, T. and Sussman, A.
LINC: The Link and Interconnection Chip.
Technical Report, Carnegie-Mellon University, Computer Science Department, May, 1984.

[12] Kung, H.T.
Let's Design Algorithms for VLSI Systems.
In *Proceedings of Conference on Very Large Scale Integration: Architecture, Design, Fabrication*, pages 65-90. California Institute of Technology, January, 1979.
Also available as a CMU Computer Science Department technical report, September 1979.

[13] Kung, H.T.
Systolic Algorithms for the CMU Warp Processor.
In *Proceedings of the Seventh International Conference on Pattern Recognition*, pages 570-577. International Association for Pattern Recognition, 1984.

[14] Kung, H.T. and Leiserson, C.E.
Systolic Arrays (for VLSI).
In Duff, I. S. and Stewart, G. W. (editors), *Sparse Matrix Proceedings 1978*, pages 256-282. Society for Industrial and Applied Mathematics, 1979.

[15] Kung, H.T. and Webb, J.A.
Global Operations on the CMU Warp Machine.
In *Proceedings of 1985 AIAA Computers in Aerospace V Conference*, pages 209-218. American Institute of Aeronautics and Astronautics, October, 1985.

[16] Kung, H. T. and Webb, J. A.
Mapping Image Processing Operations onto a Linear Systolic Machine.
Distributed Computing 1, 1986.

[17] Preparata, F.P. and Shamos, M.I.
Computational Geometry: In Introduction.
Springer-Verlag, New York, 1985.

[18] Rabiner, L.R. and Gold, B.
Theory and Application of Digital Signal Processing.
Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

[19] Rosenfeld, A.
Iterative methods in image analysis.
In *Proceedings of the IEEE Computer Society Conference on Pattern Recognition and Image Processing*, pages 14-18. International Association for Pattern Recognition, 1977.

[20] Rosenfeld, A., Hummel, R. A., and Zucker, S. W.
Scene labelling by relaxation operations.
IEEE Trans. on Systems, Man, and Cybernetics SMC-6:420-433, June, 1976.

[21] Thompson, C.D. and Kung, H.T.
Sorting on a Mesh-Connected Parallel Computer.
Communications of the ACM 20(4):263-271, April, 1977.

[22] Weiser, U. and Davis, A.
A Wavefront Notation Tool for VLSI Array Design.
In Kung, H.T., Sproull, R.F., and Steele, G.L., Jr. (editors), *VLSI Systems and Computations*, pages 226-234. Computer Science Department, Carnegie-Mellon University, Computer Science Press, Inc., October, 1981.

[23] Woo, B., Lin, L. and Ware, F.
A High-Speed 32 Bit IEEE Floating-Point Chip Set for Digital Signal Processing.
In *Proceedings of 1984 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 16.6.1-16.6.4. 1984.